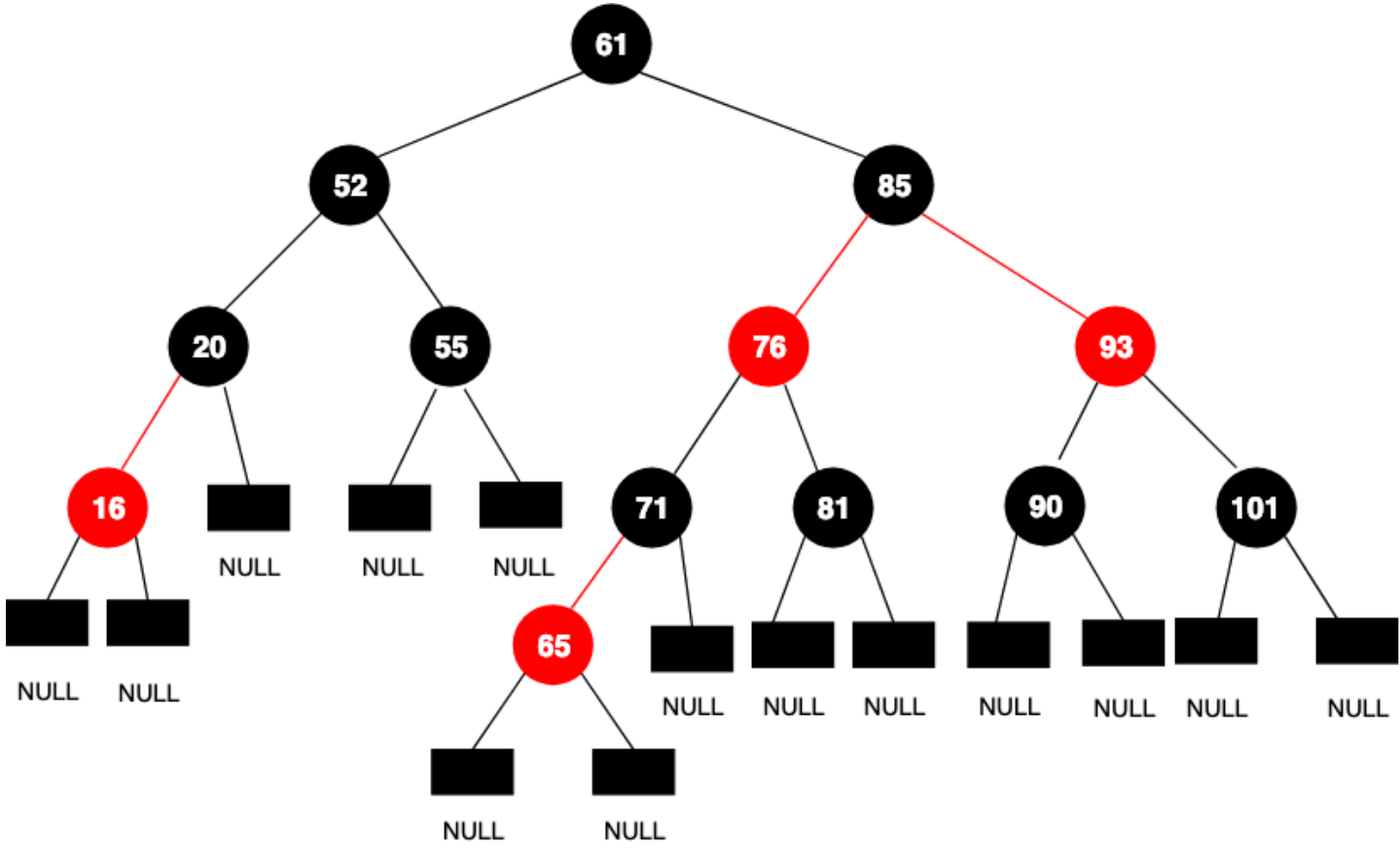
Red-black trees

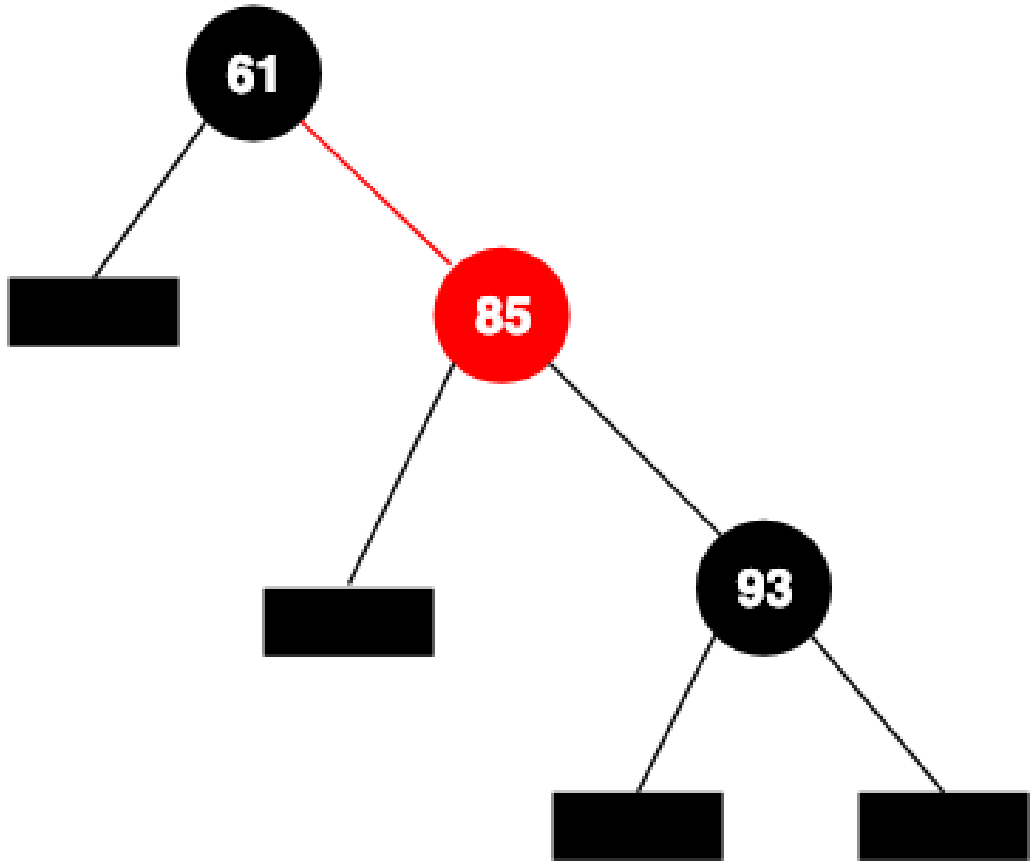
Red-black tree

- A red-black tree is a binary search tree with following properties
- 1) every node is colored either red or black
  - Null nodes are black
- 2) the root is black
- 3) if a node is red, its children must be black
- 4) every path from a node to a null pointer must contain same number of black nodes



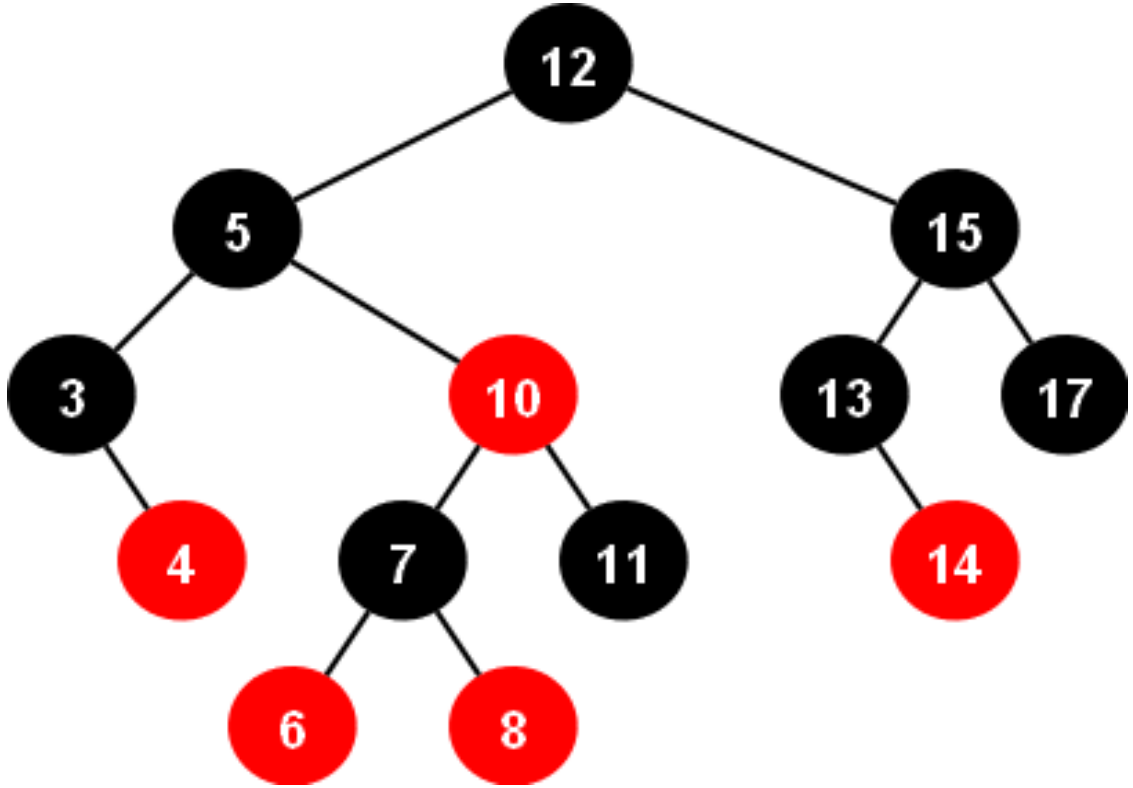
Is this a red-black tree?

- No. searching for 55 leads to leftmost null node (black) which passes only through one black node (61, root), while both of 93's children (null nodes) can only be reached by passing through *two* black nodes (61 and 93)
- Violates property 4: **every path from a node to a null pointer must contain same number of black nodes**



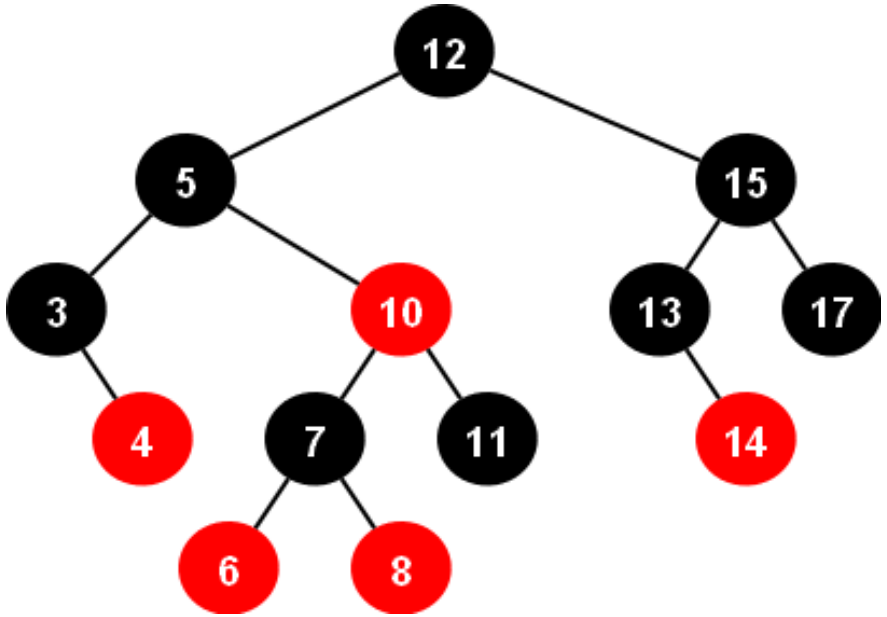
Red-black trees

- Consequence of coloring rules is that height of a red-black tree is at most $2\log(N+1)$
  - Check lemma 13.1 from algorithms textbook for proof
- $\Rightarrow$ searching is guaranteed to be logarithmic time (as with AVL trees)
- Challenge is to insert/delete while maintaining coloring rules



Insertion

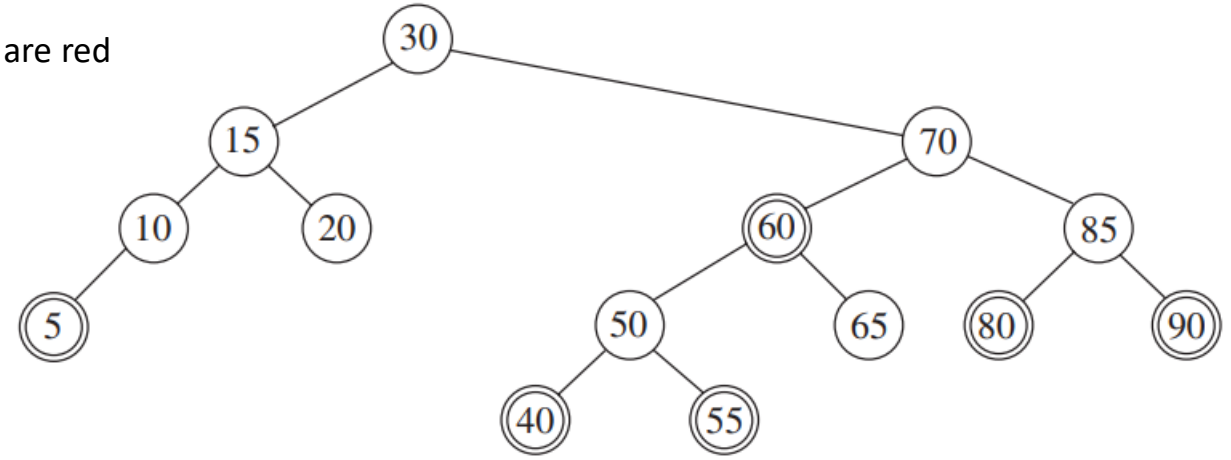
- New item will be a leaf (as with all BST insertions)
- If new item is colored black, it will violate property 4, thus new node must be colored red
  - **Property 4:** every path from node to nullptr must contain same number of black nodes
- If parent of new red node is black, we are done
- If parent of new red node is also red, we have violated property 3
  - **Property 3:** if a node is red, it's children must be black
  - In this case, we need to adjust the tree in order to ensure condition 3 is enforced (while also being sure not to violate condition 4)
  - Basic operations to adjust tree are *color changes* and *tree rotations*



Nodes with double circles are red

Insertion

- Several cases to consider when parent of newly inserted node is red:
- 1) sibling of parent (uncle) is black
  - Example: inserting 3 or 8, but *not* 99
    - Null nodes are black
- Let X be newly added leaf, let P be its parent, S be the sibling of P, and G be the grandparent.
  - Only X and P are red in this case, G is black (property 3).
- Figure 12.10 shows how we can rotate tree for case where P is a left child (there is a symmetric case)
- First case (top) corresponds to single rotation between P and G
  - Swap colors of parent and grandparent after rotation
- Second case corresponds to double rotation, first between X and P, and then between X and G
  - Swap colors of grandparent and newly inserted node after rotation
- In both cases, subtree's new root is colored black, and we removed the possibility of two consecutive red nodes. Also, number of black nodes on paths to A, B, and C is unchanged



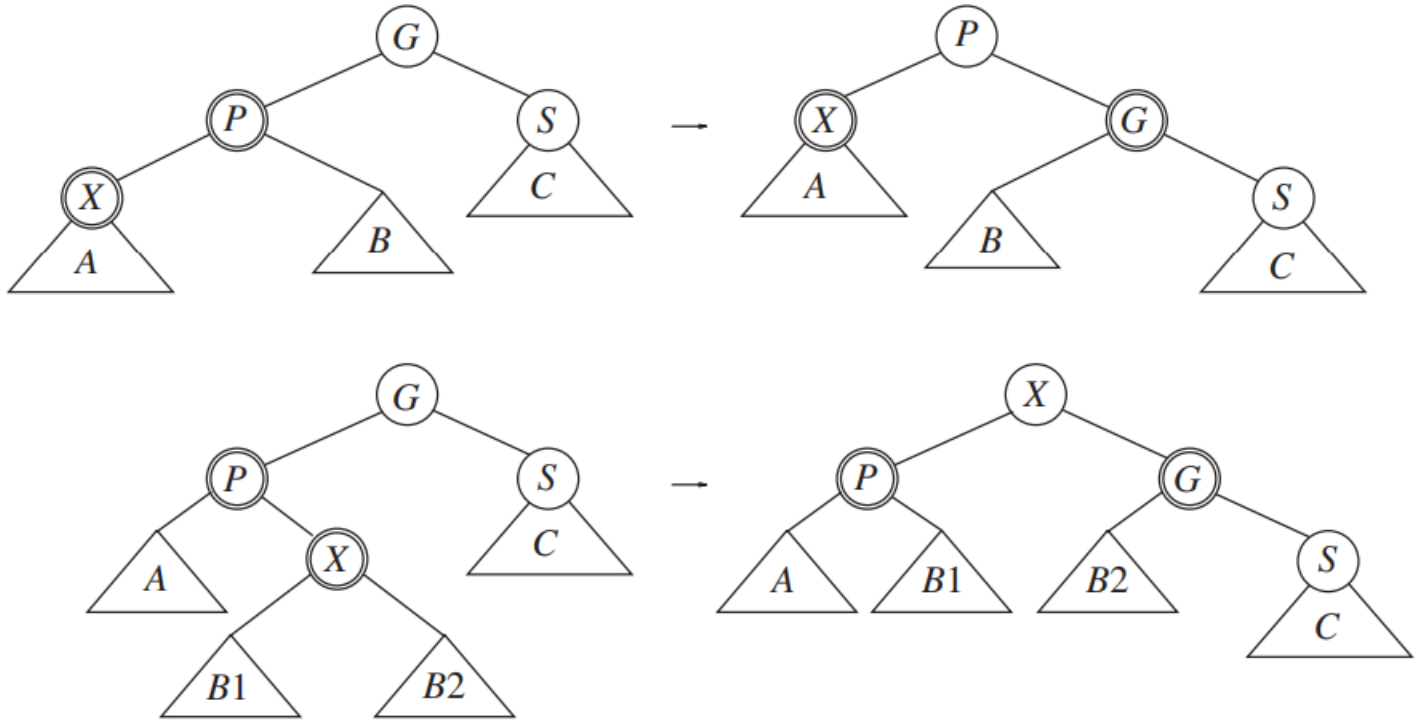
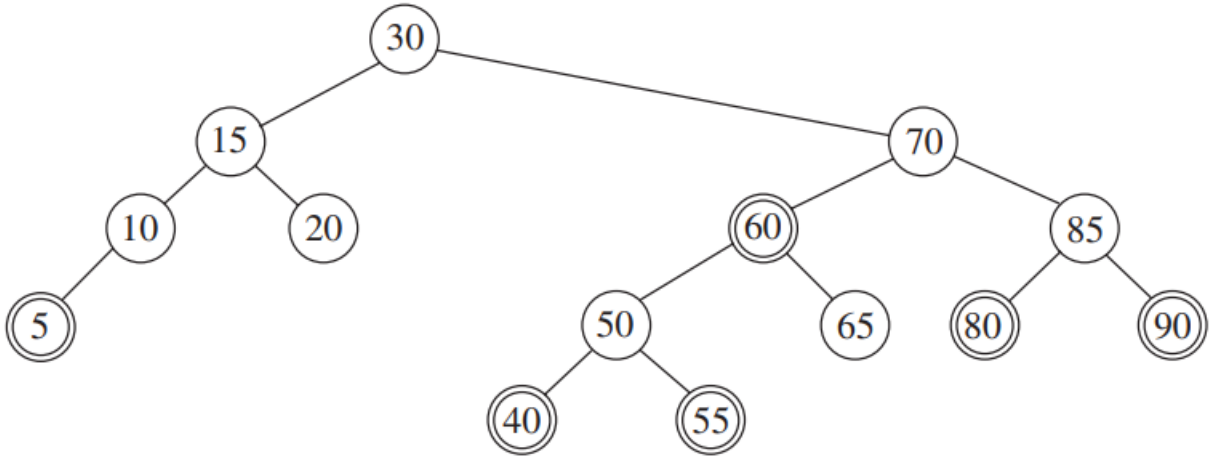


Figure 12.10 *Zig* rotation and *zig-zag* rotation work if S is black

Insertion

- What if S (sibling of newly inserted node's parent) is red, as in the case when we attempt to insert 79 into the tree?
- In this case, initially there is one black node on the path from the subtree's root to C, but after the rotation there must still be only one black node



Top-down red-black trees (insertion)

- Need a way to fix case when S is red
- Can apply a top-down procedure to the tree to guarantee that S won't be red
- On the way down, when we encounter a node X with two red children, we make X red and the two children black
  - If X is root, we change it back to black immediately after color flip to maintain property 2
- If X's parent P is red, this color flip will introduce a violation of property 3, but this can be fixed by rotations in figure 12.10
  - We are now sure that $X's$ sibling is not red, due to the color flip, so rotations in 12.10 will work fine

Figure 12.11 Color flip: only if X's parent is red do we continue with a rotation

Top-down red-black trees (insertion)

- Example: inserting 45
- On the way down the tree, we encounter node 50, which has two red children, thus we perform a color flip, making 50 red, and 40, 55 black.
- now, 50 and 60 are both red, so we perform a single rotation between 60 and 70, making 60 the black root of 30's right subtree, and 70 and 50 both red
- We continue down, performing an identical action if we see other nodes on the path that contain two red children
- Reaching the leaf, we insert 45 as a red node, and since parent is black, we are done
- Red-black tree is well balanced as can be seen in 12.12, and is faster than AVL tree for insertion

Before inserting 45

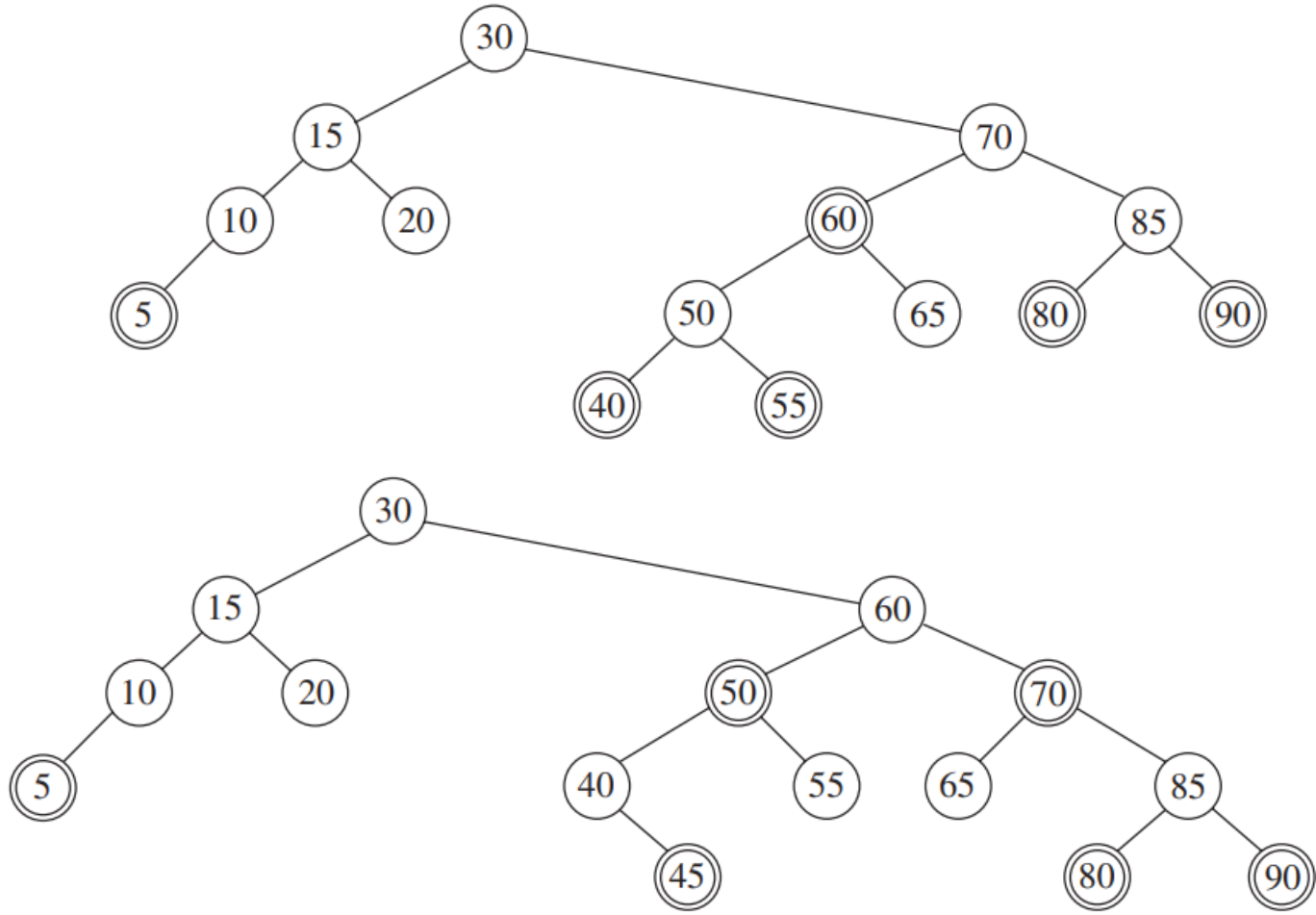


Figure 12.12 Insertion of 45 into Figure 12.9

Deletion

- Deletion always boils down to deleting a node with a single child or no children (a)
- To remove v with null-child w, w proceed as follows
- Let r be sibling of w and x be parent of v
- Remove nodes v and w and make r a child of x
- If v was red (and hence r was black) or v was black (and hence r was red), we are done (b)
- If v is black AND r is black, we assign r a fictitious 'double-black' color, introducing a color violation (c)
- A double black in the tree must be remedied on a case-by-case basis

a)

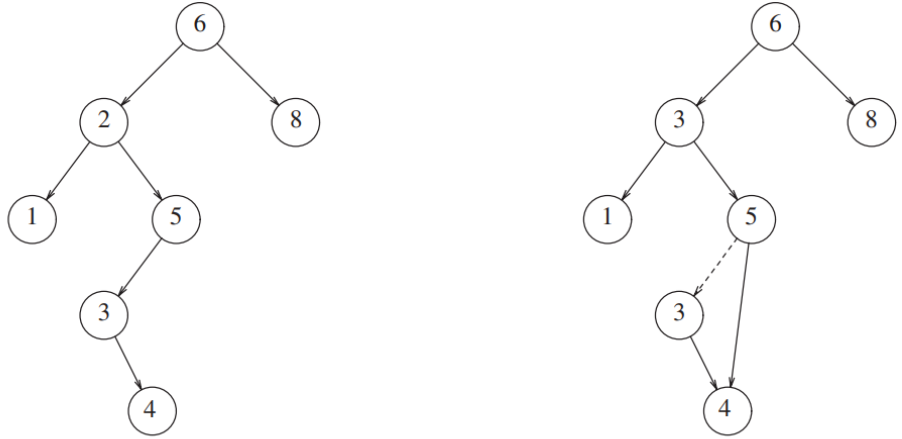
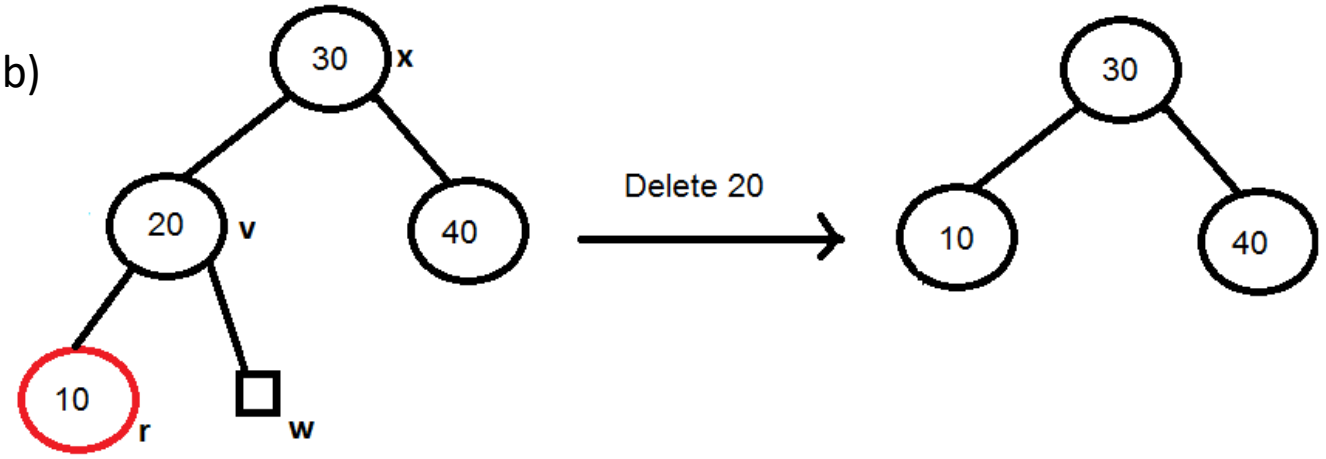
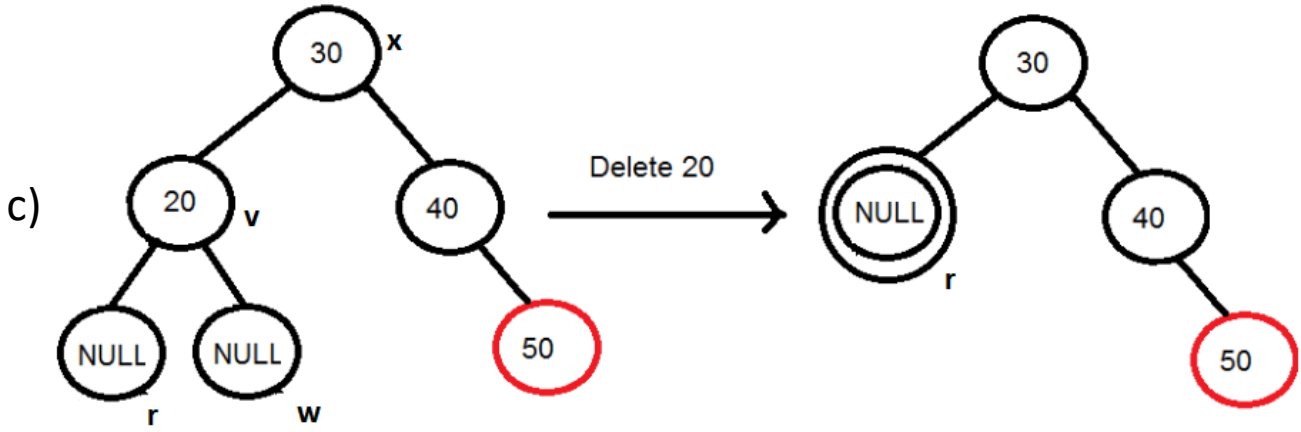


Figure 4.25 Deletion of a node (2) with two children, before and after

b)

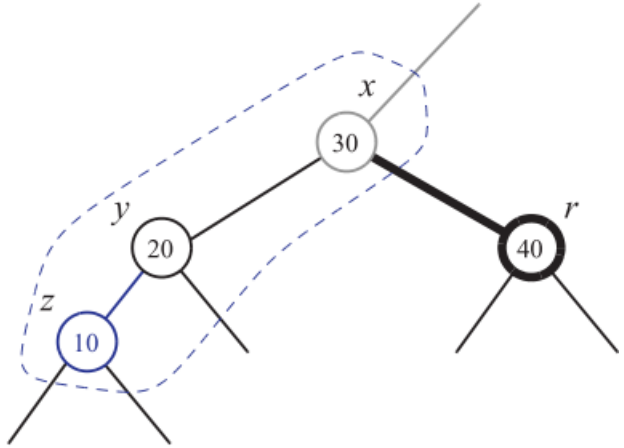


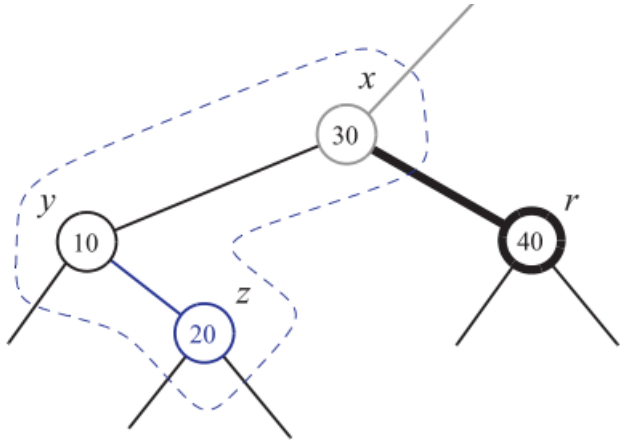
c)

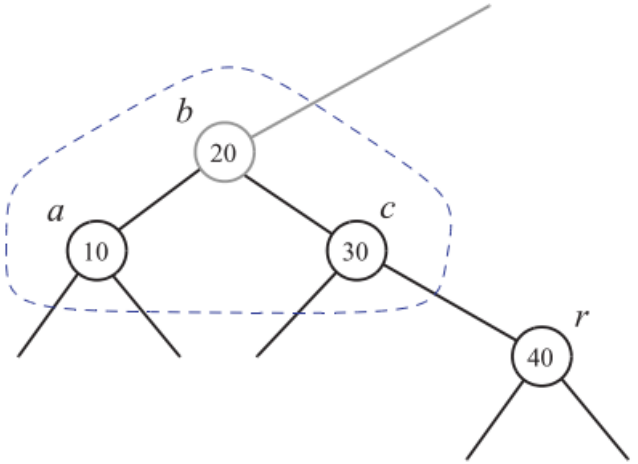


Deletion case1:

- Case 1: sibling y of r is black and has a red child z
- Perform a *trinode restructuring* by means of operation `restructure(z)`
  - This is a rotation, describes both single and double
- Restructure(z) takes node z, it's parent y, and grandparent x, labels them temporarily left-to-right as a,b,c and then replaces x with node labeled b, making it the parent of the other two
- We then color a and c black, give b the former color of x, and color r black
- This removes the double-black problem, and hence at most one restructuring is performed in this case

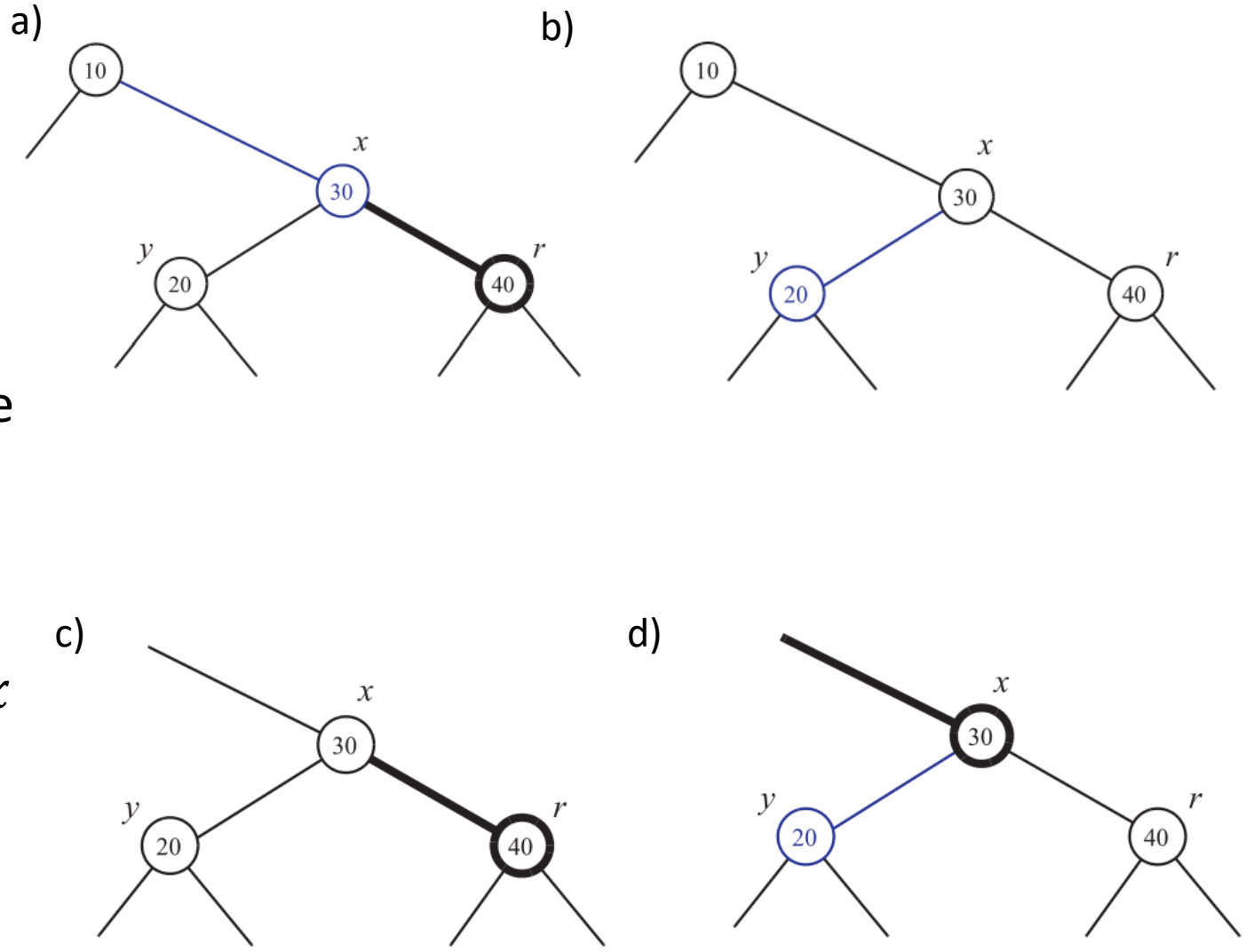






Deletion case 2

- Case 2: the sibling y of r is black and both children of y are black
- Do a recoloring to resolve this case
- color r black, color y red, and if x is red, we color it black (a,b)
- If x is not red, we color it double black, and the double-black problem reappears at the parent x of r (c,d)
- This recoloring either eliminates the double-black problem, or propagates it into the parent of the current node. We then repeat a consideration of these three cases at the parent



Deletion case 3

- Case 3: the sibling y of r is red
- In this case, perform an *adjustment* operation:
- If y is the right child of x, let z be the right child of y, otherwise, let z be the left child of y
- Execute the trinode restructuring operation `restructure(z)`, making y the parent of x
- Color y black and x red
- After this adjustment, the sibling of r is black, and either case 1 or case 2 will apply, with different meanings for x and y
  - If case 2 appears, the double-black problem cannot reappear, so to complete case 3 we will need only one more application of case 1 or 2 and then we are done